C++Builder 2006. 222 gotowe rozwiązania

Autor: Jacek Matulewski
ISBN: 83-246-0395-6
Format: B5, stron: 496

Firma Borland, producent cenionych środowisk programistycznych, połączyła swoje trzy najpopularniejsze produkty: C++ Builder, Delphi i C# Builder, tworząc pakiet Borland Developer Studio 2006. Jednym z jego elementów jest najnowsza wersja C++ Builder, oznaczona symbolem 2006, która nadal pozostaje doskonałym narzędziem do tworzenia aplikacji dla platformy Win32. C++ Builder 2006 pozwala na korzystanie z biblioteki VCL, wywołań WinAPI i funkcji systemu Windows.

Książka „C++ Builder 2006. 222 gotowe rozwiązania" to zbiór porad związanych z tworzeniem aplikacji dla systemu Windows. Wśród tytułowych 222 rozwiązań znajdziesz sposoby wykorzystywania komponentów z biblioteki VCL, elementów interfejsu programistycznego Windows (WinAPI), a także komunikatów systemowych oraz technologii COM, OLE i ActiveX. Dowiesz się, jak tworzyć biblioteki DLL i jak z nich korzystać, oraz nauczysz się projektować wspomaganą sprzętowo grafikę 3D za pomocą biblioteki OpenGL.

- Korzystanie ze środowiska C++ Builder 2006
- Projektowanie okien aplikacji
- Operacje na rejestrze systemowym
- Tworzenie wygaszaczy ekranu
- Kontrolowanie działania aplikacji za pomocą funkcji WinAPI
- Obsługa połączeń sieciowych
- Tworzenie bibliotek DLL
- Tworzenie grafiki i animacji 3D

Wykorzystaj w pracy gotowe rozwiązania, sprawdzone przez najlepszych programistów

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 11.
OpenGL

OpenGL to biblioteka funkcji realizujących operacje graficzne pozwalające na budowanie trójwymiarowych scen (zbiorów figur), oświetlanie ich, dodawanie różnego typu efektów zwiększających wrażenie realności oraz na ich prezentację na ekranie. O OpenGL można myśleć jak o interfejsie programistycznym (API) karty graficznej. Lub odwrotnie, jak o standardzie (zbiorze poleceń), który jest implementowany przez większość kart graficznych. Tak czy inaczej — renderowanie (przygotowywanie obrazu przestrzennej sceny na płaskim ekranie) wspomagane jest sprzętowo przez procesor karty graficznej (GPU), dzięki czemu jest bardzo wydajne i nie obciąża przy tym głównego procesora komputera (CPU). To pozwala na generowanie wielu scen w ciągu sekundy i ich płynną animację.

OpenGL, podobnie jak konkurencyjny Direct3D (składnik DirectX odpowiedzialny za grafikę trójwymiarową), może służyć do tworzenia gier, programów CAD czy programów do wizualizacji naukowych. W tych ostatnich zastosowaniach dobrym przykładem są programy do prezentacji niezwykle skomplikowanych struktur przestrzennych białek, co bez technik 3D byłoby trudne do wyobrażenia. Z rynku gier OpenGL został niestety wyparty przez Direct3D i to pomimo powszechnej opinii, że OpenGL jest równie wydajny, a jednocześnie bardziej elegancki i wygodniejszy w użyciu. To ostatnie twierdzenie przestało być tak oczywiste od momentu wprowadzenia wersji 8 DirectX, a szczególnie w stosunku do DirectX.NET, który jest zorientowany obiektowo. Warto podkreślić zasadniczą różnicę między DirectX i OpenGL. Ten pierwszy jest własnością Microsoftu i jest rozwijany wyłącznie przez tę korporację na potrzeby jednego systemu operacyjnego — Windows, podczas gdy OpenGL jest standardem otwartym, dostępnym na wielu platformach, implementowanym przez wielu producentów kart graficznych[1] i otwartym na rozszerzenia. Jego rozwój nadzorowany jest przez ARB — konsorcjum firm, wśród których w chwili obecnej znajdują się m.in. ATI, NVidia, 3DLabs, SGI (pierwotny twórca tej biblioteki), Sun, Apple, IBM, Dell i Intel. A więc sami giganci. Nie ma już między nimi Microsoftu, pomimo że był jedną z korporacji-założycieli. Więcej informacji na temat OpenGL, jego specyfikacji, historii, ARB i samym API można znaleźć na jego oficjalnej stronie *http://www.opengl.org/*. Warto również zajrzeć na stronę twórcy biblioteki SGI: *http://www.sgi.com/products/software/opengl/*. Poza tym

[1] Istnieje również implementacja OpenGL 1.1, przygotowana przez Microsoft, która jest dołączana do Windows.

w internecie jest bardzo dużo stron zawierających opisy, poradniki i zbiory przykładów, które łatwo odnaleźć. Są one łatwo rozpoznawane na pierwszy rzut oka, dzięki obecnemu na nich logo OpenGL (rysunek 11.1).

Rysunek 11.1
Oficjalne logo OpenGL

Zanim rozpoczniemy zabawę, chciałbym uprzedzić, że celem tego rozdziału nie jest systematyczne wprowadzenie do OpenGL. Na to potrzeba osobnych książek, a nie rozdziałów[2]. Moim zamiarem jest jedynie zainteresowanie Czytelnika możliwościami tej biblioteki, przełamanie obawy przed projektowaniem aplikacji wykorzystującej grafikę trójwymiarową i pokazanie, że OpenGL jest stosunkowo łatwy do nauczenia. Postaram się to udowodnić w serii dwudziestu sześciu ćwiczeń, które zaczniemy od rysowania trójkątów i figur przestrzennych, potem przejdziemy do obracania ich i przesuwania, następnie omówimy oświetlenie, a zakończymy na teksturowaniu kwadryk. Postaram się przy tym, aby atrakcyjność przykładów nie została stłumiona przez nadmiernie teoretyczny komentarz, ale tego nie da się zawsze uniknąć.

Rysowanie figur w przestrzeni 3D

Projekt 194. Inicjacja grafiki OpenGL w aplikacji projektowanej w środowisku C++Builder

Rozpoczniemy od przystosowania aplikacji projektowanej w środowisku C++Builder do współpracy z biblioteką OpenGL. Jest to krok dość żmudny, ale na szczęście wystarczy zrobić go tylko raz; w kolejnych projektach możemy już korzystać z gotowego szablonu.

1. Tworzymy nowy projekt *VCL Forms Application — C++Builder*. W widoku projektowania powiększamy formę.
2. Przechodzimy do kodu źródłowego nagłówka *Unit1.h* i importujemy pliki nagłówkowe *gl.h* i *glu.h* (listing 11.1). Drugi z tych plików będzie tak naprawdę potrzebny dopiero w dalszych projektach.
3. W klasie `TForm1` deklarujemy dwie prywatne metody pomocnicze i dwa prywatne pola:

[2] Jednym z ciekawszych kompendiów wiedzy o OpenGL jest *OpenGL. Księga eksperta (Wydanie III)* napisana przez Richarda S. Wrighta i Benjamina Lipchaka (Helion 2005).

Listing 11.1. *Pełny kod nagłówka Unit1.h*

```
//---------------------------------------------------------------------------

#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

#include <gl.h>
#include <glu.h>

//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:   // IDE-managed Components
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
private:       // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
public:        // User declarations
   __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Przechowanie uchwytów do kontekstu renderowania jest konieczne, aby możliwe było usunięcie go przy zamknięciu okna.

4. Definicję pierwszej metody pokazuje listing 11.2.

Listing 11.2. *Implementacja metody inicjującej aplikację OpenGL*

```
bool TForm1::GL_UstalFormatPikseli(HDC uchwytDC)
{
   PIXELFORMATDESCRIPTOR opisFormatuPikseli;
   ZeroMemory(&opisFormatuPikseli,sizeof(opisFormatuPikseli));
   opisFormatuPikseli.nVersion=1;
   opisFormatuPikseli.dwFlags=PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW |
   PFD_DOUBLEBUFFER; //w oknie, podwojne buforowanie
   opisFormatuPikseli.iPixelType=PFD_TYPE_RGBA; //typ koloru RGB
   opisFormatuPikseli.cColorBits=32; //jakosc kolorów 4 bajty
   opisFormatuPikseli.cDepthBits=16; //glebokosc bufora Z (z-buffer)
   opisFormatuPikseli.iLayerType=PFD_MAIN_PLANE;

   int formatPikseli=ChoosePixelFormat(uchwytDC,&opisFormatuPikseli);
   if (formatPikseli==0) return false;
```

```
    if (SetPixelFormat(uchwytDC,formatPikseli,&opisFormatuPikseli)!=true) return
    false;
    return true;
}
```

5. A drugiej — listing 11.3:

Listing 11.3. *Ustawienia dotyczące „filmowanej" sceny*

```
void TForm1::GL_UstawienieSceny()
{
    glViewport(0,0,ClientWidth,ClientHeight); //okno OpenGL = wnetrze formy
    (domyslnie)

    //ustawienie punktu projekcji
    glMatrixMode(GL_PROJECTION); //przełączenie na macierz projekcji
    glLoadIdentity();
    //left,right,bottom,top,znear,zfar (clipping)
    glFrustum(-0.1, 0.1, -0.1, 0.1, 0.3, 25.0); //mnozenie macierzy rzutowania przez
    macierz perspektywy - ustalanie frustum
    glMatrixMode(GL_MODELVIEW); //powrót do macierzy widoku modelu
    glEnable(GL_DEPTH_TEST); //z-buffer aktywny = ukrywanie niewidocznych
    powierzchni
}
```

6. Zmieniamy własność `BorderStyle` formy na `bsSingle` oraz `BorderIcons,biMaximize` na `false`.

7. Tworzymy metodę zdarzeniową do `OnCreate` formy i budujemy w niej kontekst OpenGL związany z bieżącym oknem (listing 11.4):

Listing 11.4. *Konieczne jest pobranie i przechowywanie uchwytu kontekstu graficznego okna — jest on potrzebny zarówno do renderowania, jak i przy zamykaniu aplikacji do zwolnienia zasobów używanych przez OpenGL*

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    //biezace okno staje sie oknem OpenGL
    uchwytDC=GetDC(Handle);
    if (!GL_UstalFormatPikseli(uchwytDC)) ShowMessage("Nie udało się ustalić formatu
    pikseli");
    uchwytRC=wglCreateContext(uchwytDC);
    if (uchwytRC==NULL) ShowMessage("Nie udało się pobrać uchwytu kontekstu
    grafiki");
    if (!wglMakeCurrent(uchwytDC,uchwytRC)) ShowMessage("Inicjacja grafiki OpenGL
    nie powiodła się");
    GL_UstawienieSceny();
    Caption=(AnsiString)"OpenGL "+(char*)glGetString(GL_VERSION);
}
```

8. Musimy pamiętać o usunięciu kontekstu OpenGL (kontekstu renderowania) przy zamknięciu okna. W tym celu tworzymy metodę zdarzeniową do `OnClose` formy i umieszczamy w niej polecenia z listingu 11.5:

Listing 11.5. *Zwalnianie zasobów zarezerwowanych podczas uruchamiania aplikacji*

```
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   wglMakeCurrent(NULL,NULL);
   wglDeleteContext(uchwytRC);
   ReleaseDC(Handle,uchwytDC);
   PostQuitMessage(0);
}
```

Po wykonaniu metody `FormCreate` główne okno aplikacji jest przygotowane do rysowania za pomocą OpenGL. W metodzie tej odczytywany jest uchwyt do kontekstu urządzenia odpowiedzialnego za przygotowywanie grafiki w oknie aplikacji (ang. *display device context*; funkcja `GetDC`) i tworzony jest kontekst związany z przygotowywaniem grafiki za pomocą biblioteki OpenGL (`wglCreateContext`), która do rysowania wykorzystywać będzie to okno (za to przypisanie odpowiada funkcja `wglMakeCurrent`). Jest to dość zawiłe, więc nie będę przezwyciężał swojej i Czytelnika zdrowej niechęci przed głębszym wnikaniem w to zagadnienie. Ważniejsze jest moim zdaniem, abyśmy dobrze zrozumieli ramy, w jakich działa OpenGL, jak inicjowana jest jej maszyna stanów[3], która przechowuje ustawienia sceny, za co odpowiedzialna jest metoda `GL_UstawienieSceny`.

Grafika trójwymiarowa, podobnie jak mechanika klasyczna, opiera się na korzystaniu z macierzy 3×3, które opisują różnego rodzaju przekształcenia w przestrzeni. Za pomocą takiej macierzy możemy zapisać przesunięcie, obrót oraz skalowanie figur, a z ich złożeń (iloczynów macierzy) zbudować dowolne przekształcenia[4]. W rzeczywistości macierze przekształceń używane w OpenGL mają rozmiar 4×4, gdzie dodatkowa współrzędna wektora opisującego punkt odgrywa rolę współczynnika skalowania (tu nie użyjemy go ani razu, więc nie będę więcej o nim wspominał). Większości przekształceń odpowiadają wygodne w użyciu funkcje (podstawowy zbiór tych funkcji zebrany został w tabeli 11.1), które budują odpowiednią macierz i mnożą przez nią bieżącą **macierz model-widok**. Czym jest macierz model-widok? To macierz zbierająca wszystkie przekształcenia[5] oryginalnego układu punktów zdefiniowanych na scenie (modelu) i transformująca je do postaci widzianej przez kamerę. Domyślnie macierz model-widok jest jednostkowa, co oznacza brak przekształceń — taką ją też zostawia metoda `GL_UstawienieSceny`.

[3] W rozdziale 9. wspomniałem, że biblioteki DLL mogą być nie tylko zbiorem funkcji, ale również posiadać stan przechowywanych w jej zmiennych globalnych. Maszyna stanu OpenGL to właśnie zbiór zmiennych biblioteki OpenGL przechowujących ustawienia dotyczące przygotowywania grafiki 3D, oświetlenia itp.

[4] Nie wymieniłem odbicia względem osi i punktu, bo OpenGL nie posiada osobnych funkcji, które realizowałyby te przekształcenia.

[5] Należy pamiętać, że iloczyn dwóch macierzy 4×4 jest nadal macierzą 4×4, a więc jedną macierzą można opisywać dowolne złożenie przekształceń.

Tabela 11.1. *Funkcje OpenGL mają wiele odmian przyjmujących różne typy argumentów*

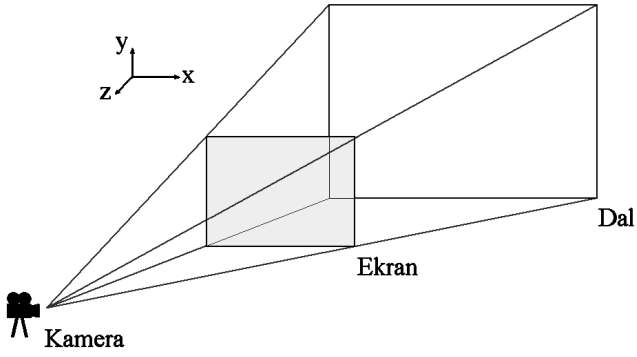
Rodzaj przekształcenia	Funkcja (nazwa bez przyrostków)	Najczęściej wykorzystywana wersja
Przesunięcie (translacja)	`glTranslate`	`glTranslatef`
Obrót	`glRotate`	`glRotatef`
Skalowanie	`glScale`	`glScalef`

Pomnożenie oryginalnych współrzędnych sceny przez macierz model-widok to jednak dopiero pierwszy etap przygotowywania obrazu, jaki widzimy na ekranie. Kolejnym etapem jest rzutowanie. Związana jest z nim druga macierz nazywana **macierzą rzutowania**. Polecenie `glMatrixMode(GL_PROJECTION);`, które znajduje się w metodzie `GL_UstawienieSceny`, zmienia bieżącą macierz (tj. macierz, której dotyczą funkcje z tabeli 11.1) z macierzy model-widok identyfikowanej przez stałą `GL_MODELVIEW` na macierz rzutowania (stała `GL_PROJECTION`). Jeżeli chcemy, możemy ją przesunąć, obrócić czy powiększyć. Jednak nie ma to większego sensu. Zamiast tego zmienimy ją tak, aby przekształcała scenę w taki sposób, że rzeczy odległe są mniejsze od tych, które znajdują się bliżej kamery. Do tego właśnie służy macierz rzutowania — może wprowadzać rzut perspektywiczny. Rzutowanie to nie zachowuje oczywiście pierwotnych odległości pomiędzy punktami sceny. Alternatywą jest rzut izometryczny (prostopadły, ortogonalny), w którym przedmioty znajdujące się dalej od kamery są tej samej wielkości, co bliższe niej. Ten sposób rzutowania może wydawać się nierealistyczny i wobec tego nieprzydatny, ale znają go np. wszyscy miłośnicy gier fabularnych, w których sprawdza się doskonale. Co robimy z macierzą rzutowania? Modyfikujemy ją za pomocą funkcji `glFrustum`, która ustala własności **frustum**. Jeszcze jedno nowe pojęcie? Frustum to po prostu obszar, na którym budowana jest scena — wycinek przestrzeni, który „filmuje" kamera, ograniczony od strony kamery przez ekran. Przedmioty znajdujące się poza frustum nie są renderowane. W zależności od stosowanego typu rzutowania frustum może mieć kształt prostopadłościanu, którego jedną ze ścian jest ekran (rzutowanie ortogonalne), lub, jak jest w powyższym przykładzie, widocznego na rysunku 11.2 ostrosłupa o podstawie prostokąta ze ściętym czubkiem (rzutowanie perspektywiczne). Widoczna na listingu 11.3 funkcja `glFrustum` mnoży jednostkową macierz rzutowania[6] przez macierz perspektywy wygenerowaną na podstawie argumentów funkcji, generując w efekcie rzut perspektywiczny:

```
glFrustum(lewo,prawo,dół,góra,blisko,daleko);
```

Argumenty wyznaczają położenie ekranu w układzie odniesienia, w którym kamera (niektórzy wolą mówić „oko") znajduje się w środku układu współrzędnych (punkt `(0,0,0)`). Oznacza to, że dwa ostatnie argumenty wyznaczające najbliższy i najdalszy plan powinny być większe od zera. W powyższym przykładzie mają one wartości odpowiednio `0.3` i `25.0`. Należy pamiętać, że ich różnica wyznacza stopień zniekształcenia obrazu — jak wspomniałem, w rzucie perspektywicznym kąty i odległości nie są zachowane. Jeżeli *`blisko`* będzie bliskie zera, to figury będą się szybko zmniejszać w miarę oddalania się od kamery i szybko przestaną być widoczne. Można się przekonać o tym, zmieniając na chwilę ten parametr z `0.3` na `0.03` i uruchamiając aplikację.

[6] Macierz jednostkowa została przygotowana funkcją `glLoadIdentity`.



Rysunek 11.2. *Frustum w rzucie perspektywicznym*

Przestrzeń ograniczana przez frustum opisana jest za pomocą prawoskrętnego układu współrzędnych, którego osie OX i OY skierowane są odpowiednio w prawo i do góry względem płaszczyzny ekranu, natomiast oś OZ skierowana jest od dali do kamery[7]. Domyślne położenie środka układu współrzędnych, względem którego ustalane jest położenie figur, zgodne jest z położeniem kamery. Zatem położenie płaszczyzny ekranu na osi OZ w naszym przykładzie to `-0.3`, a najdalszego planu `-25.0` (obie są ujemne). Środek układu współrzędnych znajduje się w ten sposób poza frustum. Rysowanie sceny zaczniemy od jej przesunięcia o `-10`, a więc w kierunku dali, aby środek współrzędnych był widoczny na ekranie (znajdował się wewnątrz frustum). Po zmodyfikowaniu macierzy rzutowania przywracamy macierz model-widok jako bieżącą macierz (polecenie `glMatrixMode(GL_MODELVIEW);`). To na niej wykonywać będziemy wszystkie pozostałe przekształcenia w programie. W istocie macierz rzutowania po jej zainicjowaniu rzadko jest zmieniana. Metoda `GL_UstawienieSceny` kończy się poleceniem `glEnable (GL_DEPTH_TEST);`, którym włączamy **bufor głębi**, odpowiedzialny za kontrolowanie odległości figur od kamery i ich odpowiednie wzajemne przesłanianie.

Podsumowując, przypomnijmy, z jakimi macierzami mamy do czynienia w OpenGL. Po pierwsze jest to macierz model-widok. Jeżeli na scenie narysowana jest postać człowieka stojącego na płaszczyźnie OXZ (głowa skierowana jest w kierunku dodatnim osi OY), a kamera wisi nad nim i jest skierowana w dół, to przekształcenie zadane przez macierz model-widok spowoduje, że pierwotny układ współrzędnych zostanie przesunięty tak, że jego początek będzie znajdował się w położeniu kamery oraz będzie obrócony wokół osi OX o 90 stopni przeciwnie do kierunku ruchu wskazówek zegara (patrząc w stronę dodatniej półosi OX). **Macierz model-widok mówi zatem, jak zmienić układ odniesienia sceny na układ odniesienia kamery.** Drugą macierzą jest macierz rzutowania, która może wprowadzić do obrazu perspektywę. Końcowym etapem jest przygotowanie obrazu dwuwymiarowego (tzw. **przekształcenie widoku**), który wyświetlany jest na ekranie. Przekształcenie widoku możemy wyobrazić sobie bardzo łatwo: gdybyśmy stanęli przed oknem i, trzymając głowę nieruchomo, obrysowali flamastrem na szybie kontury wszystkich budynków widzianych za oknem — powstałby dwuwymiarowy rzut trójwymiarowej przestrzeni. Mniej więcej na tym polega

[7] Proszę zwrócić uwagę, że zwykle oś OY skierowana jest w dół ekranu. OpenGL jest w tym względzie zgodne z intuicją uzyskaną w szkole podstawowej. Podobnie jest z osią OZ, która jest skierowana „do nas", a nie jak w Direct3D „od nas". Dzięki temu układ współrzędnych jest prawoskrętny, co pozwala na stosowanie bez dodatkowych przekształceń wzorów z podręczników do geometrii.

przekształcenie widoku (nie opisuje go już oczywiście macierz 3×3). Te trzy przekształcenia tworzą tak zwany **potok**, który obejmuje cały proces przygotowywania obrazu przez OpenGL.

Projekt 195. Rysowanie figury płaskiej (trójkąta). Podwójne buforowanie

1. Deklarujemy metodę `RysujScene` zgodnie ze wzorem z listingu 11.6.

Listing 11.6. *Deklaracja metody renderującej*

```
class TForm1 : public TForm
{
__published:   // IDE-managed Components
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
private:      // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   void __fastcall RysujScene();
public:       // User declarations
   __fastcall TForm1(TComponent* Owner);
};
```

2. Następnie definiujemy ją zgodnie z listingiem 11.7. Trójkąt rysowany jest w pobliżu punktu `(0,0,0)`, ale przesuniętego o `10` w dal[8].

Listing 11.7. *Definicja metody renderującej*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
```

[8] Zauważmy, że nie ma znaczenia, czy przekształceniom poddana zostaje scena, czy kamera. Przesunięcie sceny w lewo jest całkowicie równoważne przesunięciu kamery w prawo — to tylko kwestia wyboru układu odniesienia.

```
    glVertex3f(0, y0, z0); //gorny
    //koniec rysowania figury
    glEnd();

    //Z bufora na ekran
    glFlush();
    SwapBuffers(uchwytDC);
}
```

Przyrostek nazwy wskazuje na ilość i typ przeciążonych funkcji; np. `glVertex3f` oznacza, że przyjmowane są trzy argumenty typu `float`. Funkcja `glVertex` ma znacznie więcej odmian (w sumie około dwudziestu pięciu). W dokumentacji dołączonej do BDS informacji o funkcjach OpenGL należy szukać pod hasłem bez przyrostka, a więc `glVertex`, a nie `glVertex3f`.

3. Za pomocą inspektora obiektów tworzymy metodę związaną ze zdarzeniem `OnPaint` formy i umieszczamy w niej wywołanie metody `RysujScene` (listing 11.8). Dzięki temu rysunek sceny będzie odświeżany.

Listing 11.8. *Do zainicjowania renderowania sceny wykorzystujemy zdarzenie OnPaint formy*

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
    RysujScene();
}
```

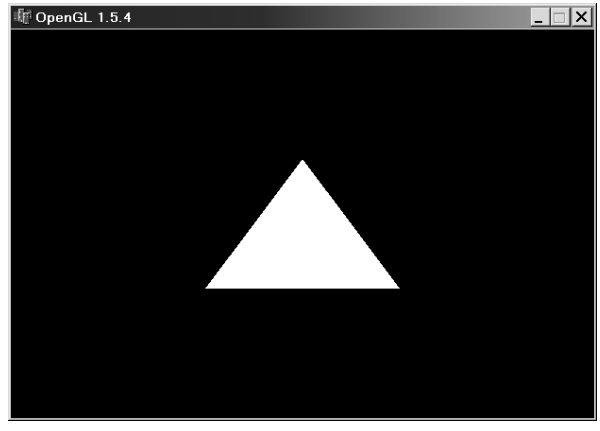
Metodę `RysujScene` otwiera polecenie przypisujące macierzy model-widok (będącej macierzą bieżącą) macierz jednostkową (funkcja `glLoadIdentity`), która oznacza, że układ odniesienia sceny i kamery są jednakowo ustawione. Nie trwa to jednak długo, bo znajdujące się w następnej linii polecenie `glTranslatef(0.0,0.0,-10.0);` rozsuwa scenę i kamerę o `10.0` jednostek. Dzięki temu rozsunięciu trójkąt, który rysujemy w pobliżu środka układu współrzędnych, znajduje się wewnątrz frustum i możemy zobaczyć go na ekranie.

Rysowanie trójkąta to sekwencja poleceń rozpoczynająca się od funkcji `glBegin`, a kończąca się wywołaniem funkcji `glEnd`. Argumentem funkcji `glBegin` jest typ figury (punkt, linia, trójkąt, czworokąt lub wielokąt oraz złożenia trójkątów). Między `glBegin` a `glEnd` powinny znaleźć się przede wszystkim wywołania funkcji `glVertex`, które definiują **werteksy**, czyli punkty w trójwymiarowej przestrzeni, a praktyce wierzchołki figur. Zadeklarowany w `glBegin` typ figury wyznacza sposób interpretacji werteksów przez bibliotekę OpenGL. Jeżeli argumentem `glBegin` jest `GL_POINTS`, to każdy werteks jest reprezentowany przez osobno rysowany punkt. Z kolei `GL_LINES` powoduje łączenie werteksów w pary, między którymi rysowany jest odcinek. Stała `GL_LINE_STRIP` powoduje rysowanie łamanej między wszystkimi zdefiniowanymi werteksami. Dla `GL_TRIANGLES` i `GL_QUADS` werteksy grupowane są odpowiednio w trójki i czwórki, a na ekranie pojawia się trójkąt lub czworokąt. Zatem definicja jednego trójkąta to sekwencja wywołań funkcji `glBegin(GL_TRIANGLES)`, trzykrotnego `glVertex` i `glEnd`. Wywołań `glVertex` mogłoby być także sześć, dziewięć, dwanaście itd., a wówczas zdefiniowalibyśmy nie jeden, ale dwa, trzy, cztery i więcej trójkątów.

Projekt 196. Poprawianie geometrii frustum

Zwróćmy uwagę, że w efekcie wykonania poleceń z poprzedniego ćwiczenia otrzymaliśmy trójkąt równoboczny (por. rysunek 11.3). Jednak definiowaliśmy trójkąt o podstawie `2*x0` i wysokości `2*y0`, a więc o podstawie `2` i wysokości `2`. Długości jego lewego i prawego boku powinny więc być równe w przybliżeniu `2.23`. A ponieważ trójkąt ten leży na płaszczyźnie równoległej do ekranu, to wynika z tego, że nie powinniśmy uzyskać niczego, co wygląda na trójkąt równoboczny (rysunek 11.3); trójkąt powinien być lekko „wydłużony". Co się wobec tego z nim stało? Okazuje się, że z trójkątem wszystko jest w porządku. Przyczyna deformacji obrazu jest inna: określając frustum określiliśmy wielkość najbliższego planu (ekranu) na rozciągającą się od `-0.1` do `0.1` w kierunkach X i Y. Z tego wynika, że każdy plan sceny ma kształt kwadratu. A przecież okno widoczne na rysunku 11.3 nie ma takiego kształtu — jest rozciągnięty w poziomie. I to rozciągnięcie spowodowało zniekształcenie naszego trójkąta. Proporcje okna (a w trybie pełnoekranowym proporcje ekranu) należy uwzględnić przy definiowaniu frustum (listing 11.9).

Rysunek 11.3. *Pierwszy trójkąt*

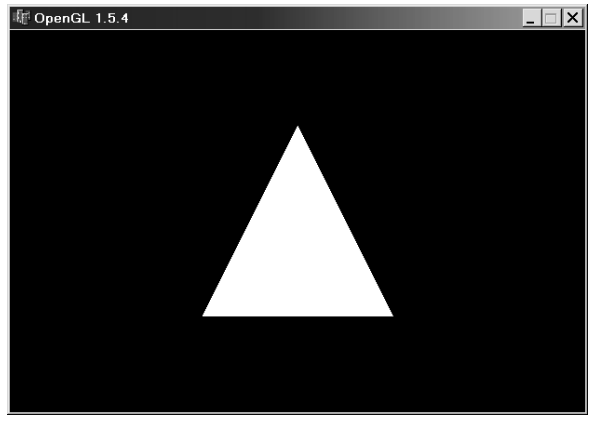


Listing 11.9. *Definiując frustum, uwzględniamy proporcję okna*

```
void TForm1::GL_UstawienieSceny()
{
   glViewport(0,0,ClientWidth,ClientHeight); //okno OpenGL = wnetrze formy
   (domyslnie)

   //ustawienie punktu projekcji
   glMatrixMode(GL_PROJECTION); //macierz projekcji
   glLoadIdentity();
   //left,right,bottom,top,znear,zfar (clipping)
   float wsp=ClientHeight/(float)ClientWidth;
   glFrustum(-0.1, 0.1, wsp*-0.1, wsp*0.1, 0.3, 25.0); //mnozenie macierzy przez
   macierz perspektywy - ustalanie piramidy frustum
   glMatrixMode(GL_MODELVIEW); //powrot do macierzy widoku modelu
   glEnable(GL_DEPTH_TEST); //z-buffer aktywny = ukrywanie niewidocznych trojkatow
!!!
}
```

Na rysunku 11.4 widoczny jest trójkąt po skorygowaniu proporcji frustum.

Rysunek 11.4. *Trójkąt — tym razem wiernie odwzorowany na ekranie*

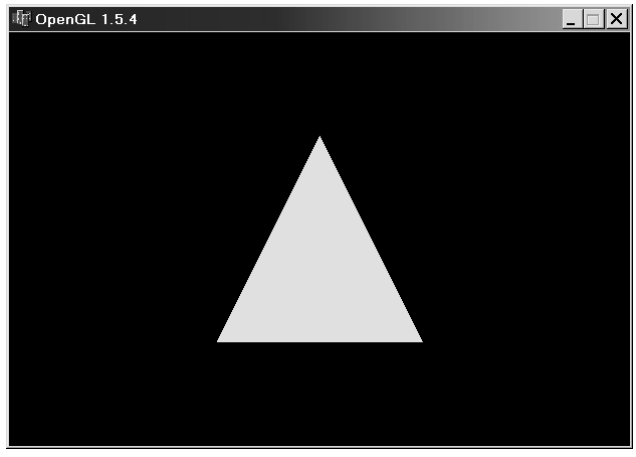


Projekt 197. Kolor

Do powyższej funkcji dodajemy wywołanie funkcji `glColor`, a dokładnie jej wersji `glColor3ub` z argumentami `255`, `255` i `0`, które opisują odpowiednio czerwoną, zieloną i niebieską składową koloru (listing 11.10). W efekcie płaszczyzna trójkąta stanie się żółta, co dobrze byłoby widać na rysunku 11.5, gdyby książka miała kolorowe ilustracje.

Listing 11.10. *Czarno-biały świat to żadna zabawa*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu
   glColor3ub(255,255,0); //zolty
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny
   //koniec rysowania figury
   glEnd();

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```



Rysunek 11.5. *Zmiana koloru trójkąta*

Zagadnienie kolorów to znacznie poważniejsza sprawa niż wynika z powyższego listingu. Jest ona w OpenGL tak ściśle związana z oświetleniem, że na zabawę kolorami musimy poczekać, aż włączymy mechanizm oświetlenia.

Projekt 198. Rysowanie figury przestrzennej (ostrosłupa)

Jak wspominałem, między funkcjami `glBegin(GL_TRIANGLES)` i `glEnd` można umieścić nie tylko trzy definicję werteksów, ale dowolną wielokrotność tej liczby. Dodajmy wobec tego jeszcze trzy trójki, które określą trzy dodatkowe trójkąty (listing 11.11). Całość powinna utworzyć ostrosłup. Każda ściana ostrosłupa narysowana będzie w innym kolorze:

Listing 11.11. *Rysowany wcześniej trójkąt też był figurą przestrzenną (umieszczoną w przestrzeni trójwymiarowej), tyle że… płaską*

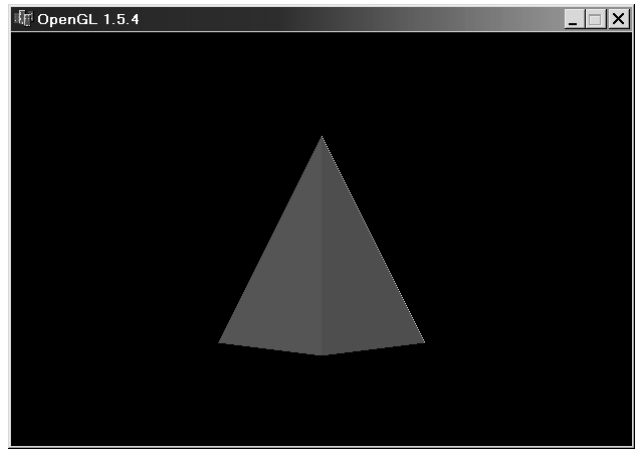
```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   glColor3ub(255,255,0); //zolty
   glVertex3f( -x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
```

```
    glVertex3f(0, y0, z0); //gorny

    //podstawa
    glColor3ub(0,255,0); //zielony
    glVertex3f( -x0, -y0, z0); //dolny lewy
    glVertex3f(x0, -y0, z0); //dolny prawy
    glVertex3f(0, -y0, 2*z0); //dolny przedni

    //lewa
    glColor3ub(255,0,0); //czerwony
    glVertex3f( -x0, -y0, z0); //dolny lewy
    glVertex3f(0, -y0, 2*z0); //dolny przedni
    glVertex3f(0, y0, z0); //gorny

    //prawa
    glColor3ub(0,0,255); //niebieski
    glVertex3f(x0, -y0, z0); //dolny prawy
    glVertex3f(0, -y0, 2*z0); //dolny przedni
    glVertex3f(0, y0, z0); //gorny

    //koniec rysowania figury
    glEnd();

    //Z bufora na ekran
    glFlush();
    SwapBuffers(uchwytDC);
}
```

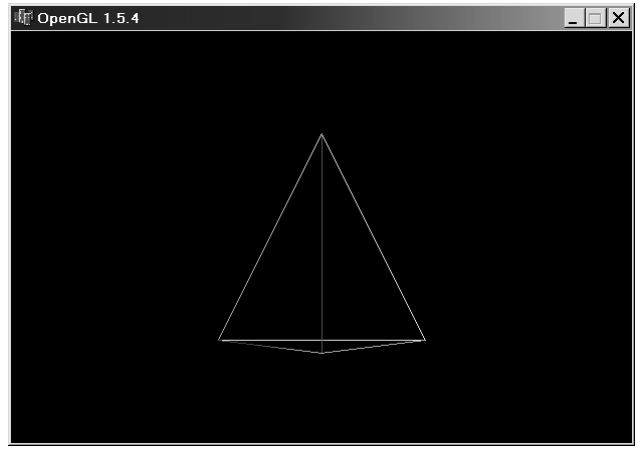
Zamiast rysować cztery trójkąty (12 wierzchołków do przekształcania przez macierze model-widok i rzutowania), można było posłużyć się wstęgami lub wachlarzami trójkątów (jako argumentów funkcji `glBegin` należy wówczas użyć zamiast `GL_TRIANGLES` odpowiednio `GL_TRIANGLE_STRIP` lub `GL_TRIANGLE_FAN`). Wówczas kolejne trójkąty wyznaczane są nie przez nową trójkę werteksów, a przez jeden dodatkowy werteks i jeden z boków poprzednio zdefiniowanego trójkąta. W ten sposób zmniejsza się ilość wierzchołków, co w przypadku dużych figur zbudowanych z wielu werteksów może znacząco przyspieszyć ich przekształcanie w potoku.

Natomiast aby zamiast „pełnego" ostrosłupa (rysunek 11.6) rysować jedynie jego szkielet, należy w `glBegin` użyć argumentu `GL_LINE_LOOP` (zob. rysunek 11.7).

Kolejną ważną sprawą związaną z rysowaniem figur płaskich jest tzw. **nawinięcie**. Kolejność werteksów wyznacza bowiem przód i tył figury. Jeżeli staniemy „za" figurą (tj. patrzymy w kierunku jej przodu), to wierzchołki powinny być zdefiniowane zgodnie z kierunkiem ruchu wskazówek zegara. Tym samym jeżeli patrzymy na trójkąt narysowany na ekranie i jego wierzchołki nawinięte są w kierunku przeciwnym do kierunku ruchu wskazówek zegara, to trójkąt skierowany jest przodem do nas. Dlaczego jest to ważne? Ponieważ można uniknąć rysowania figur, które skierowane są np. tyłem do kamery. Bardzo ogranicza to wysiłek włożony przez kartę graficzną w przygotowanie obrazu. Ale czy sprawdzanie, czy figura ułożona jest przodem, czy tyłem do kamery, nie jest kosztowniejsze niż rysowanie obu jej stron? Owszem, ale wyobraźmy sobie zamkniętą figurę przestrzenną, np. nasz ostrosłup. Jeżeli jego ściany będą tak nawinięte, że wszystkie będą skierowane przodem na zewnątrz, to możemy mieć pewność, że rysowanie



Rysunek 11.6. *Każdą figurę przestrzenną można zbudować z trójkątów*



Rysunek 11.7. *Wszystkie możliwe argumenty funkcji glBegin znajdzie Czytelnik w dokumentacji MSDN dołączonej do C++Buildera*

„tyłów" nie będzie potrzebne. Są jednak od tej sytuacji wyjątki. Możemy na przykład dopuszczać sytuację, w której kamera wnika do wnętrza figury. Wówczas, jeżeli chcemy widzieć jej wnętrze, a nie ma po prostu znikać, musimy rysować figury z obu stron. Inna możliwość, i to jest właśnie nasz przypadek, jest taka, że zamierzamy uczynić jedną lub kilka ścian częściowo przezroczystych. A przez takie okno będzie można oczywiście również zajrzeć do wnętrza figury, a więc wówczas także konieczne jest rysowanie figur z obu stron.

Projekt 199. Wyodrębnienie metody rysującej figurę

Wygodnie jest umieścić zbiór poleceń rysujących figurę w osobnej metodzie lub funkcji. Umożliwi to wielokrotne użycie tych funkcji i w ten sposób łatwe powielanie figury (por. projekt 204.).

1. Definiujemy metodę `RysujOstroslup` i umieszczamy w niej polecenia od `glBegin` do `glEnd z metody RysujScene` (listing 11.12).

Listing 11.12. *Metoda definiująca wierzchołki ostrosłupa*

```
void __fastcall TForm1::RysujOstroslup(float x0,float y0,float z0)
{
   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);
   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   glColor3ub(255,255,0); //zolty
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, y0, z0); //gorny

   //podstawa
   glColor3ub(0,255,0); //zielony
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni

   //lewa
   glColor3ub(255,0,0); //czerwony
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
   glVertex3f(0, y0, z0); //gorny

   //prawa
   glColor3ub(0,0,255); //niebieski
   glVertex3f(x0, -y0, z0); //dolny prawy
   glVertex3f(0, -y0, 2*z0); //dolny przedni
   glVertex3f(0, y0, z0); //gorny

   //koniec rysowania figury
   glEnd();
}
```

2. Do definicji klasy dodajemy odpowiednią deklarację nowej metody.
3. Następnie modyfikujemy metodę RysujScene, zastępując w niej sekwencję poleceń od `glBegin` do `glEnd` przez wywołanie `RysujOstroslup` (listing 11.13).

Listing 11.13. *Taka zmiana zdecydowanie zwiększa czytelność metody renderującej*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10
```

```
    //rysowanie ostroslupa
    RysujOstroslup(x0,y0,z0);

    //Z bufora na ekran
    glFlush();
    SwapBuffers(uchwytDC);
}
```

Projekt 200. Obroty obiektów na scenie. Ruch kamery i ruch aktorów

Czas zająć się przekształceniami macierzy model-widok. Tu pojawia się jednak zagadnienie kolejności przekształceń i poleceń rysujących nowe figury. Musimy wiedzieć jedną ważną rzecz: **to, co zostało narysowane, nie jest już przekształcane**. Zatem jeżeli chcemy obracać ostrosłupem, to funkcje `glRotate` muszą znaleźć się przed `RysujOstroslup`, bo inaczej obracane będą figury zdefiniowane potem, ale ostrosłup zostanie nieruchomy.

Ta zasada pozwala na uniezależnienie ruchu poszczególnych przedmiotów znajdujących się na scenie (aktorów), a także, a może przede wszystkim, na oddzielenie ruchu kamery od ruchu aktorów. Jak dobrze wiemy, nie ma zasadniczej różnicy między ruchem kamery, a ruchem sceny, szczególnie jeżeli na scenie nie ma niczego, co nasz umysł mógłby identyfikować jako nieruchome (np. jakiś rodzaj podłoża). Zresztą nawet wówczas stwierdzenie, który element się porusza, a który spoczywa, jest umowne[9]. Wyobraźmy sobie jednak metodę renderującą, w której wpierw rysowana jest płaszczyzna, następnie wykonywane jest przekształcenie translacji (przesunięcia) i wówczas rysowany jest ostrosłup. Wyglądałoby to tak, jakby ostrosłup przesuwał się po nieruchomej (tj. narysowanej przed przesunięciem) powierzchni — tak w naturalny sposób scenę tę interpretowałby nasz umysł. Jeżeli dodatkowe przekształcenie umieścimy przed narysowaniem owej powierzchni, to dotyczyć one będą zarówno niej, jak i poruszającego się po niej ostrosłupa. Wyglądać to będzie, jak ruch kamery, która filmuje ruch ostrosłupa po powierzchni.

Z tego wynika, że można umownie odróżnić ruch kamery od ruchu obiektów na scenie. Podział ten zależy od momentu, w którym rysowana jest część sceny, którą uznajemy za nieruchomą (owa powierzchnia). Wcześniejsze przekształcenia to ruch kamery, późniejsze — ruchy aktorów. Ogólny schemat metody renderującej powinien być zatem zgodny z przedstawionym w tabeli 11.2.

I tak dalej. Etapy 7. i 8. mogą się oczywiście powtarzać. Aktor nie tylko może nie być nieruchomy (tzn. może poruszać się po scenie), ale może również zmieniać kształt. Do tego względnego ruchu trójkątów, z których zbudowany jest aktor, stosuje się jednak ta sama zasada, co do ruchu aktorów po scenie.

[9] Każdy chyba pamięta gag z filmu Monty Pythona, w którym to nie pociąg odjeżdżał, ale peron z osobą machającą białą chusteczką.

Tabela 11.2. *Kolejność czynności wykonywanych w metodzie renderującej*

Etap	Czynności
1. Inicjacja	Macierz model-widok ustalana na jednostkową
2. Przedmioty trwale związane z kamerą (np. broń bohatera widoczna w grach FPP/FPS)	Rysowanie figur
3. Ruch kamery	Przekształcenia macierzy model-widok
4. Rysowanie nieruchomej części sceny	Rysowanie figur
5. Ruch pierwszego aktora	Przekształcenia macierzy model-widok
6. Rysowanie pierwszego aktora	Rysowanie figur
7. Ruch drugiego aktora	Przekształcenia macierzy model-widok
8. Rysowanie drugiego aktora	Rysowanie figur

Aby móc obracać nasz ostrosłup za pomocą klawiszy sterowania kursorem, musimy wykonać następujące czynności:

1. Definiujemy dwa pola `Theta` i `Phi` typu `Single`, przechowujące kąty określające położenie ostrosłupa (listing 11.14).

Listing 11.14. *Definiujemy dwa nowe pola*

```
class TForm1 : public TForm
{
__published:   // IDE-managed Components
   void __fastcall FormCreate(TObject *Sender);
   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
   void __fastcall FormPaint(TObject *Sender);
private:      // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   void __fastcall RysujScene();
   void __fastcall RysujOstroslup(float x0,float y0,float z0);
   float Phi, Theta;
public:       // User declarations
   __fastcall TForm1(TComponent* Owner);
};
```

2. W konstruktorze formy inicjujemy nowe pola zerami (listing 11.15).

Listing 11.15. *Inicjujemy oba kąty zerami*

```
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
   Phi=0; Theta=0;
}
```

3. Do metody renderującej dodajemy polecenia obracające macierz model-widok o kąty zapamiętane w polach `Phi` i `Theta` (listing 11.16). Pierwsze wyznacza kąt obrotu wokół osi OY, a drugie — wokół osi OX[10].

Listing 11.16. *W każdej renderowanej scenie położenie figury wyznaczone jest przez pola Phi i Theta*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //obroty
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

4. Zmieniamy własność `KeyPreview` formy na `true`.

5. Za pomocą inspektora tworzymy metodę związaną z `OnKeyDown` formy (listing 11.17). Będzie ona odgrywała rolę naszego pulpitu kontrolnego.

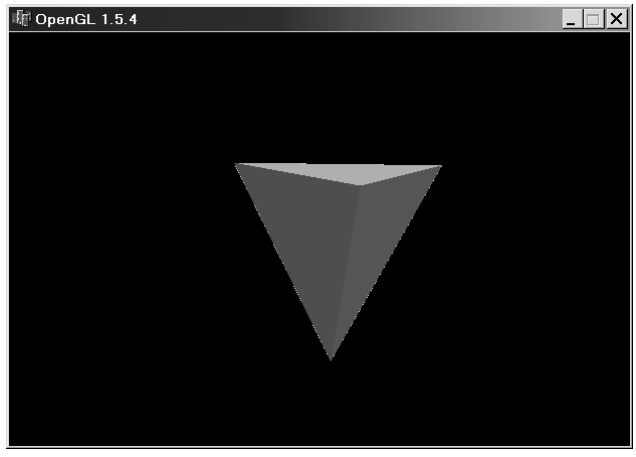
Listing 11.17. *„Strzałki" pozwalają na kontrolowanie wartości kątów, o jakie obracany jest ostrosłup*

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift)
{
   switch (Key)
   {
      case VK_ESCAPE: Close(); break;
      //obroty
      case VK_LEFT:   Phi-=3; break;
      case VK_RIGHT:  Phi+=3; break;
      case VK_UP :    Theta-=3; break;
      case VK_DOWN:   Theta+=3; break;
```

[10] Kolejność obrotów ma znaczenie! Tak samo jak kolejność mnożenia przez macierze. Wystarczy wziąć do ręki niniejszą książkę i obrócić ją wzdłuż grzbietu, a potem wzdłuż innego wybranego boku. Następnie wróćmy do pozycji wyjściowej i wykonajmy oba obroty w zmienionej kolejności. W obu przypadkach uzyskamy inne ułożenie książki.

```
    }

    RysujScene();
}
```

Korzystając z „klawiszy strzałek”, możemy obracać nasz ostrosłup i obejrzeć go z dowolnej strony (rysunek 11.8).



Rysunek 11.8. *Teraz możemy przekonać się, że figura, którą zbudowaliśmy, jest rzeczywiście trójwymiarowa*

Projekt 201. Przesunięcia obiektu

Analogicznie wygląda sprawa z przekształceniem translacji:

1. W klasie `TForm1` definiujemy jeszcze dwa pola prywatne typu `Single` o nazwach `PozycjaX`, `PozycjaY` i `PozycjaZ`. W konstruktorze ustalamy ich wartości na równe 0 (listing 11.18).

Listing 11.18. *Również te pola inicjujemy zerami*

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    Phi=0; Theta=0;
    PozycjaX=0; PozycjaY=0; PozycjaZ=0;
}
```

2. Do metody `FormPaint` dodajemy polecenie przesuwające figurę o wektor `[PozycjaX, PozycjaY, PozycjaZ]` zgodnie ze wzorem na listingu 11.19:

Listing 11.19. *Przesuwamy ostrosłup*

```
void __fastcall TForm1::RysujScene()
{
    const float x0=1.0;
    const float y0=1.0;
    const float z0=1.0;
```

```
   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //obroty
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

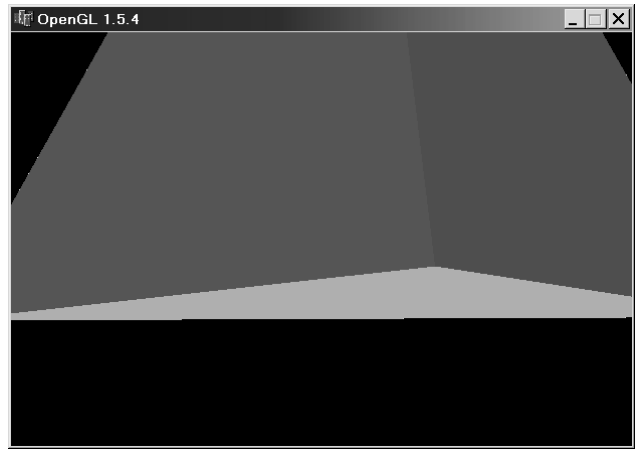
3. Rozbudowujemy metodę `FormKeyDown`, dodając do niej fragmenty wyróżnione na listingu 11.20. Dzięki temu będziemy mogli kontrolować przesunięcia ostrosłupa klawiszami sterowania kursorem przy jednoczesnym naciśnięciu klawisza *Shift* lub *Ctrl*.

Listing 11.20. *Zwiększamy ilość przycisków na pulpicie kontrolnym*

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift)
{
   if (Key==VK_ESCAPE) Close();

   //obroty
   if (Shift.Empty())
   {
      switch (Key)
      {
         case VK_LEFT:   Phi-=3; break;
         case VK_RIGHT:  Phi+=3; break;
         case VK_UP :    Theta-=3; break;
         case VK_DOWN:   Theta+=3; break;
      }
   }

   //przesuniecia w pionie i poziomie
   if (Shift.Contains(ssCtrl))
   {
      switch (Key)
      {
         case VK_LEFT:  PozycjaX-=0.1; break;
         case VK_RIGHT: PozycjaX+=0.1; break;
         case VK_UP:    PozycjaY+=0.1; break;
         case VK_DOWN:  PozycjaY-=0.1; break;
      }
   }
```

```
    //przesuniecia w poziomie i przod-tyl
    if (Shift.Contains(ssShift))
    {
       switch (Key)
       {
          case VK_LEFT:  PozycjaX-=0.1; break;
          case VK_RIGHT: PozycjaX+=0.1; break;
          case VK_UP:    PozycjaZ-=0.1; break;
          case VK_DOWN:  PozycjaZ+=0.1; break;
       }
    }

    RysujScene();
}
```

Po wykonaniu powyższych czynności możemy za pomocą „klawiszy strzałek" oraz klawiszy *Shift* i *Ctrl* w pełni kontrolować pozycję ostrosłupa (rysunek 11.9). Dopóki na scenie jest tylko jeden obiekt, nie ma znaczenia, czy nazwiemy to ruchem kamery, czy aktora. Nasz wzrok i tak nie ma żadnego „nieruchomego" układu odniesienia. To tak, jakbyśmy krążyli wokół innej osoby w pustej przestrzeni kosmicznej. Pytanie o to, kto wiruje wokół kogo, nie ma wówczas żadnego sensu. Lub mniej abstrakcyjny przykład — dwa pociągi stojące na sąsiednich torach. Gdy jeden z nich rusza (i nie widzimy nieruchomego podłoża) nie możemy stwierdzić który z nich jedzie, a który stoi. Sytuacja zmienia się całkowicie, gdy na scenie umieścimy coś, co nasz umysł zidentyfikuje jako nieruchome podłoże. Jednak, jak już podkreślałem wcześniej, z punktu widzenia OpenGL przekształcenia w punkcie 3. przedstawionego wyżej schematu niczym nie różnią się od przekształceń z punktów 5. i 7.



Rysunek 11.9. *Teraz naszemu ostrosłupowi możemy przyjrzeć się z bliska*

Projekt 202. Prosta animacja

Animacja to po prostu przesunięcia i obroty wykonywane zgodnie z pewnym czasowym schematem. W C++Builderze ów schemat możemy najprościej wyznaczyć za pomocą komponentu `TTimer`. Wystarczy, że co określony czas będzie on uruchamiał metodę zdarzeniową, w której zmieniane będą pola przechowujące wartości kątów i pozycji

ostrosłupa, po czym wywoływał metodę `RysujScene`. Identycznie jak przy naciskaniu klawiszy sterowania kursorem, z tym że teraz owe klawisze „naciska" sam komputer.

1. Umieszczamy na formie komponent `TTimer` z zakładki *System*.
2. Tworzymy metodę zdarzeniową do `OnTimer` (listing 11.21):

Listing 11.21. *Aby ruch był widoczny, konieczne jest oczywiście wywołanie metody RysujScene*

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
   Phi+=1;
   Theta+=0.1;
   RysujScene();
}
```

3. Zmieniamy własność `Interval` komponentu `Timer1` na `10`.
4. Do metody `FormKeyDown` dodajemy możliwość włączenia i wyłączenia animacji za pomocą klawisza *Q* (listing 11.22):

Listing 11.22. *Nowy przycisk na panelu kontrolnym*

```
void __fastcall TForm1::FormKeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift)
{
   if (Key==VK_ESCAPE) Close();

   //obroty
   if (Shift.Empty())
   {
      switch (Key)
      {
         case VK_LEFT:   Phi-=3; break;
         case VK_RIGHT:  Phi+=3; break;
         case VK_UP :    Theta-=3; break;
         case VK_DOWN:   Theta+=3; break;

         case 'q':
         case 'Q':  Timer1->Enabled=!Timer1->Enabled; break;
      }
   }

//ciag dalszy metody
```

Dzięki temu w trakcie animacji metoda `RysujScene` nie tylko jest wywoływana w reakcji na zdarzenie `OnPaint` formy, co zachodzi stosunkowo rzadko, ale przede wszystkim cyklicznie co 10 milisekund (sto razy na sekundę). To z pewnością wystarczająco często, bo płynna animacja wymaga około 50 klatek na sekundę (a w najgorszym wypadku przynajmniej 25). To może nawet zbyt wiele, bo monitory CRT (kineskopowe) rzadko pracują z częstością większą niż 80 Hz (80 „ekranów" na sekundę). Częstość odświeżania monitorów LCD, z natury wolniejszych, wynosi zazwyczaj 60 Hz.

Jeżeli nawet nieco przeholujemy z częstością renderowania sceny, to nie musimy się martwić o obciążenie głównego procesora komputera — rysowaniem grafiki OpenGL zajmuje się przecież procesor karty graficznej[11].

Projekt 203. Rysowanie osi układu współrzędnych

Narysujmy dwa układy współrzędnych: biały i zielony. W naszym przykładzie białe osie będą odgrywały rolę nieruchomej części sceny. Ustalamy, że wszystkie przekształcenia macierzy model-widok wyprzedzające narysowanie owych białych osi nazywać będziemy ruchem kamery. Wszystkie późniejsze — ruchem aktorów.

1. Definiujemy metodę `RysujOsie` rysującą osie układu współrzędnych OXYZ (listing 11.23):

Listing 11.23. *Jeżeli chcemy narysować dwa układy współrzędnych, wygodnie przygotować odpowiednią metodę, którą będziemy mogli wielokrotnie wykorzystywać*

```
procedure TForm1.RysujOsie(rozmiar :Single);
begin
glBegin(GL_LINES);
glVertex3f(0,0,0); glVertex3f(rozmiar,0,0); //OX, w prawo
glVertex3f(0,0,0); glVertex3f(0,rozmiar,0); //OY, do gory
glVertex3f(0,0,0); glVertex3f(0,0,rozmiar); //OZ, do kamery
glEnd;
end;
```

2. Metodę `RysujOsie` wywołujemy po raz pierwszy przed poleceniami wykonującymi obroty i translacje ostrosłupa, a po raz drugi już po narysowaniu ostrosłupa (listing 11.24). Pierwsze wywołanie poprzedzamy zmianą koloru na biały, a drugie — na zielony.

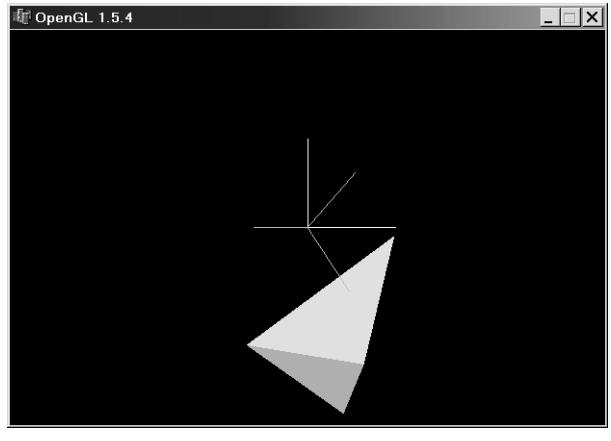
Listing 11.24. *Rysowanie nieruchomego i ruchomego układu współrzędnych*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //nieruchome osie ukladu wspolrzednych
   glColor3ub(255,255,255);
   RysujOsie(x0);
```

[11] Oczywiście, jeżeli nie mamy jakiejś archaicznej karty graficznej nie obsługującej standardu OpenGL i korzystamy z programowej emulacji.

```
    //obroty
    glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
    glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

    //przesuniecia
    glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

    //rysowanie ostroslupa
    RysujOstroslup(x0,y0,z0);

    //ruchome osie ukladu wspolrzednych
    glColor3ub(100,255,100);
    RysujOsie(x0);

    //Z bufora na ekran
    glFlush();
    SwapBuffers(uchwytDC);
}
```

Metoda `RysujOsie` wywoływana jest dwukrotnie. Pierwsze wywołanie następuje tuż po ustaleniu pozycji kamery i poprzedza obroty i przesunięcia, które interpretujemy jako ruch aktorów. Dlatego ten rysowany na biało układ współrzędnych można traktować jak związany z nieruchomą sceną (por. rysunek 11.10). Drugi raz metoda `RysujOsie` wywoływana jest tuż przed wymianą buforów, dotyczą jej zatem wszystkie przekształcenia macierzy model-widok — innymi słowy, osie związane są z rysowanymi na scenie aktorami.



Rysunek 11.10. *Rysujemy dwie pary osi współrzędnych — pierwsza związana jest z nieruchomym układem odniesienia sceny, druga — z układem związanym z ostrosłupem*

Projekt 204. Dodawanie kolejnych figur

Do pierwszego ostrosłupa dorysujemy kolejne, uzupełniając metodę `RysujScene` o pętlę wyróżnioną w listingu 11.25. Każdy z ostrosłupów obrócony jest o 90 stopni względem poprzedniego. Wszystkie będą względem siebie nieruchome, ale dodanie względnego ruchu między nimi byłoby jedynie kwestią uzmiennienia wielkości obrotu, jaki jest wykonywany przed kolejnym wywołaniem metody `RysujOstroslup`. Uzyskany efekt widoczny jest na rysunku 11.11.

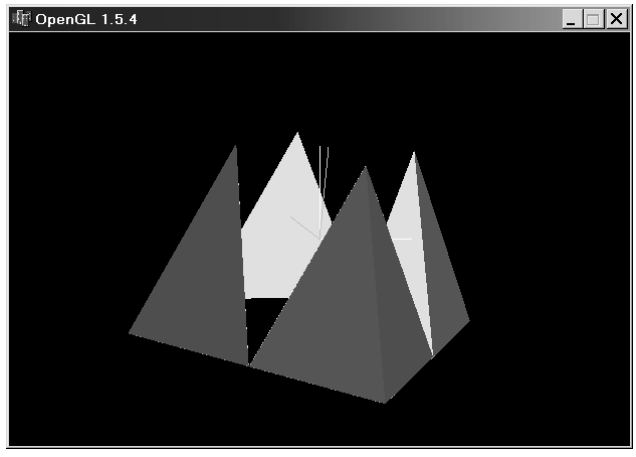
Listing 11.25. *Klonowanie ostrosłupów*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //nieruchome osie ukladu wspolrzednych
   glColor3ub(255,255,255);
   RysujOsie(x0);

   //obroty
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //kolejne ostroslupy
   for (int i=0;i<3;i++)
   {
      glRotatef(90.0, 0.0, 1.0, 0.0);
      RysujOstroslup(x0,y0,z0);
   }
   glRotatef(90.0, 0.0, 1.0, 0.0); //dopelnienie pelnego obrotu

   //ruchome osie ukladu wspolrzednych
   glColor3ub(100,255,100);
   RysujOsie(x0);

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

W kodzie umieszczonym na płycie rysowanie dodatkowych ostrosłupów można wyłączyć klawiszem *W*.

Projekt 205. Bardziej precyzyjne ustawianie kamery

Jak pamiętamy z komentarza do zadań 194. i 195., renderowanie rozpoczynamy od przesunięcia całej sceny o -10 w kierunku osi OZ, a więc o 10 w głąb frustum. Zgodnie ze schematem przedstawionym we wprowadzeniu do projektu 200. przekształcenie to znajduje



Rysunek 11.11. *Korona zbudowana z czterech ostrosłupów*

się w punkcie 3., a więc powinniśmy mówić raczej o odsunięciu kamery od sceny. Realizację tego schematu w tej chwili w metodzie `RysujScene` przedstawia tabela 11.3.

Tabela 11.3. *Schemat etapów w metodzie RysujScene*

Etap	Czynności
1. Inicjacja	Macierz model-widok ustalana na jednostkową
2. Przedmioty trwale związane z kamerą	Brak
3. Ruch kamery	Odsunięcie kamery o `10` jednostek w kierunku osi OZ
4. Rysowanie nieruchomej części sceny	Rysowanie białych osi współrzędnych
5. Ruch pierwszego aktora	Obroty i przesunięcia kontrolowane z klawiatury
6. Rysowanie pierwszego aktora	Rysowanie ostrosłupa (oraz jego klonów)
7. Ruch drugiego aktora	Brak
8. Rysowanie drugiego aktora	Rysowanie zielonych osi współrzędnych

Czynność ustawienia kamery (można sobie wyobrazić, że etap 3. obejmuje znacznie więcej przekształceń niż tylko nasze przesunięcie) można znacznie uprościć, korzystając z funkcji `gluLookAt`. Jej argumentami są trzy trójki współrzędnych. Pierwsza ustala położenie kamery. W naszym przypadku będzie to punkt `(0,0,10)`. Argumenty od czwartego do szóstego wskazują punkt, na który kamera jest skierowana. My wybraliśmy środek układu współrzędnych, czyli miejsce, w którym znajduje się nasza figura (pamiętajmy, że kamera została odsunięta na pozycję `(0,0,10)`). Ostatnia trójka argumentów określa sposób, w jaki ustawiona jest kamera, wskazując jej kierunek „do góry". My ustawiliśmy ją w kierunku osi OY, czyli tak, jak trzymalibyśmy prawdziwą kamerę, gdyby podłoże znajdowało się na płaszczyźnie OXZ (por. rysunek 11.2). Odpowiednie wywołanie funkcji `gluLookAt` pokazuje listing 11.26.

Funkcja `gluLookAt` jest zdefiniowana w bibliotece GLU (*OpenGL Utility Library*) — bibliotece zawierającej funkcje wyższego poziomu (krzywe, figury przestrzenne itp.). Więcej powiemy o niej w podrozdziale znajdującym się na końcu rozdziału. Do jej użycia potrzebny jest import nagłówka *glu.h*.

Listing 11.26. *W metodzie RysujScene zastąpiliśmy polecenie odsuwające scenę o –10 poleceniem precyzyjniej ustalającym położenie kamery*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   gluLookAt(0,0,10,  //polozenie kamery
             0,0,0,   //punkt, na ktory skierowana jest kamera
           0,1,0);  //kierunek "do gory" kamery (polaryzacja)

   //nieruchome osie ukladu wspolrzednych
   glColor3ub(255,255,255);
   RysujOsie(x0);

   //obroty
   glRotatef(Phi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(Theta, 1.0, 0.0, 0.0); //wokol OX

   //przesuniecia
   glTranslatef(PozycjaX,PozycjaY,PozycjaZ);

   //rysowanie ostroslupa
   RysujOstroslup(x0,y0,z0);

   //kolejne ostroslupy
   if (DodatkoweOstroslupy)
   {
      for (int i=0;i<3;i++)
      {
         glRotatef(90.0, 0.0, 1.0, 0.0);
         RysujOstroslup(x0,y0,z0);
      }
      glRotatef(90.0, 0.0, 1.0, 0.0); //dopelnienie pelnego obrotu
   }

   //ruchome osie ukladu wspolrzednych
   glColor3ub(100,255,100);
   RysujOsie(x0);

   //Z bufora na ekran
   glFlush();
   SwapBuffers(uchwytDC);
}
```

Projekt 206. Ruch kamery kontrolowany położeniem kursora myszy

Kontrola pozycji kamery za pomocą myszki to standardowy element aplikacji korzystających z grafiki 3D. Możliwe jest kilka podejść do tego zagadnienia — ja wybiorę może nie do końca satysfakcjonujące, ale zdecydowanie najprostsze. Co więcej, będę niekonsekwentny. W poprzednim paragrafie przedstawiłem bowiem funkcję `gluLookAt`, która służy do łatwego ustawiania kamery. Natomiast do jej obracania zastosuję poznane wcześniej funkcje `glRotate`. Tak jest po prostu wygodniej[12].

Ruch kamery technicznie realizowany jest analogicznie do przenoszenia formy za dowolny punkt (projekt 14.). Wykorzystamy do tego celu zdarzenia `OnMouseDown`, `OnMouseMove`, `OnMouseUp`.

1. Wyłączamy animację, zmieniając własność `Timer1->Enabled` na `false`.
2. Do klasy `TForm1` dodajemy trzy pola prywatne określające położenie kamery we współrzędnych sferycznych `KameraPhi`, `KameraTheta`, `KameraR`, dwa dodatkowe `tmpKameraPhi`, `tmpKameraTheta`, których zastosowanie zaraz się wyjaśni, oraz dwa pola `X0` i `Y0`, które umożliwiają mierzenie przesunięcia myszki z przyciśniętym przyciskiem (listing 11.27).

Listing 11.27. *Definiujemy potrzebne pola*

```
private:   // User declarations
   HDC uchwytDC; //uchwyt do "display device context (DC)"
   HGLRC uchwytRC; //uchwyt do "OpenGL rendering context"
   bool GL_UstalFormatPikseli(HDC uchwytDC);
   void GL_UstawienieSceny();
   void __fastcall RysujScene();
   void __fastcall RysujOstroslup(float x0,float y0,float z0);
   float Phi, Theta;
   float PozycjaX, PozycjaY, PozycjaZ;
   bool DodatkoweOstroslupy;
   void __fastcall RysujOsie(float rozmiar);
   //kamera
   int X0,Y0;
   float KameraPhi, KameraTheta, KameraR;
   float tmpKameraPhi, tmpKameraTheta;
```

3. Inicjujemy je, dodając do konstruktora instrukcje wyróżnione w listingu 11.28.

Listing 11.28. *Wartość pola KameraR (odległość kamery) kontrolować będziemy rolką myszki*

```
__fastcall TForm1::TForm1(TComponent* Owner)
   : TForm(Owner)
{
```

[12] Nie tylko wygodniej, ale również bezpieczniej. Korzystanie z `glLookAt` wymagałoby obliczania transformacji współrzędnych punktu, w którym znajduje się kamera (właśnie to robią funkcje `glRotate`). Naprawdę łatwo wówczas o błąd. A jedenaste przykazanie programisty mówi: nie pisz kodu, który ktoś napisał (i sprawdził!) za ciebie. To nie tylko wygoda, ale i unikanie błędów.

```
    Phi=0; Theta=0;
    PozycjaX=0; PozycjaY=0; PozycjaZ=0;
    DodatkoweOstroslupy=false;
    KameraR=10; KameraPhi=0; KameraTheta=0;
    X0=0; Y0=0;
}
```

4. W metodzie `RysujScene` modyfikujemy argument funkcji `gluLookAt` odpowiadający za odległość kamery od środka układu współrzędnych oraz dodajemy polecenia obrotu (listing 11.29).

Listing 11.29. *Dodane czynności należą oczywiście do etapu 3., przedstawionego w tabeli 11.3 schematu*

```
void __fastcall TForm1::RysujScene()
{
   const float x0=1.0;
   const float y0=1.0;
   const float z0=1.0;

   //Przygotowanie bufora
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity(); //macierz model-widok = macierz jednostkowa
   //glTranslatef(0.0, 0.0, -10.0); //odsuniecie calosci o 10

   //kamera
   gluLookAt(0,0,KameraR,  //polozenie kamery
        0,0,0,   //punkt, na ktory skierowana jest kamera
        0,1,0);  //kierunek "do gory" kamery (polaryzacja)
   glRotatef(KameraPhi+tmpKameraPhi, 0.0, 1.0, 0.0); //wokol OY
   glRotatef(KameraTheta+tmpKameraTheta, 1.0, 0.0, 0.0); //wokol OX

   //nieruchome osie ukladu wspolrzednych
   glColor3ub(255,255,255);
   RysujOsie(x0);

//dalsza czesc metody
```

5. Następnie tworzymy metody zdarzeniowe związane ze zdarzeniami `OnMouseDown`, `OnMouseMove`, `OnMouseUp` i umieszczamy w nich polecenia widoczne na listingu 11.30.

Listing 11.30. *Sposób kontrolowania kamery przez myszkę jest podobny do wykorzystanego w projekcie 14. do poruszania formą za dowolny punkt*

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
     TShiftState Shift, int X, int Y)
{
   X0=X;
   Y0=Y;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormMouseMove(TObject *Sender, TShiftState Shift, int X,
     int Y)
```

```
{
   //Caption="KameraPhi="+FloatToStr(KameraPhi)+",
     KameraTheta="+FloatToStr(KameraTheta);
   float czuloscMyszy=5;
   if (Shift.Contains(ssLeft))
   {
      int dX=X-X0;
      int dY=Y-Y0;
      tmpKameraPhi=dX/czuloscMyszy;
      tmpKameraTheta=dY/czuloscMyszy;
      RysujScene();
   }
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
     TShiftState Shift, int X, int Y)
{
   KameraPhi+=tmpKameraPhi;
   KameraTheta+=tmpKameraTheta;
   tmpKameraPhi=0;
   tmpKameraTheta=0;
   //kosmetyka
   if (KameraPhi>=360) KameraPhi-=360;
   if (KameraPhi<0) KameraPhi+=360;
   if (KameraTheta>=360) KameraTheta-=360;
   if (KameraTheta<0) KameraTheta+=360;
}
```

6. Ruch myszki steruje ruchem kamery na sferze wokół środka układu współrzędnych. Natomiast rolką myszki ustalać będziemy promień tej sfery, a więc odległość kamery od naszego ostrosłupa. Wykorzystamy w tym celu zdarzenie `OnMouseWheel`, do którego należy utworzyć metodę zdarzeniową z poleceniami widocznymi na listingu 11.31.

Listing 11.31. *Dodatkowo definiujemy funkcję pomocniczą sign, która ustala znak (–1, 0 lub 1) podanej liczby*

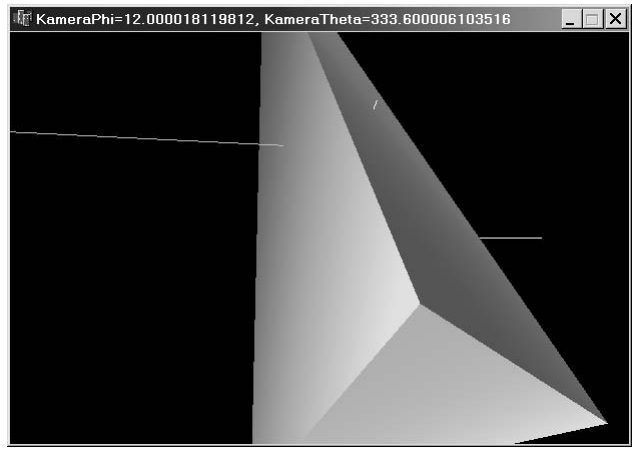
```
int sign(int argument)
{
   if (argument<0) return -1;
   if (argument>0) return 1;
   return 0;
}

void __fastcall TForm1::FormMouseWheel(TObject *Sender, TShiftState Shift,
     int WheelDelta, TPoint &MousePos, bool &Handled)
{
   const float wsp=0.1;
   //proporcjonalna zmiana pozycji wszystkich wsp. kamery
   KameraR=KameraR*(1+sign(WheelDelta)*wsp);
   RysujScene();
}
```

W momencie naciśnięcia przycisku myszy zapamiętywane jest położenie myszki (do tego wykorzystujemy pola `X0` i `Y0`). W metodzie związanej ze zdarzeniem `OnMouseMove`, jeżeli przyciśnięty jest lewy klawisz myszy, obliczana jest wielkość przesunięcia myszy od momentu naciśnięcia przycisku myszy i na tej podstawie obliczane są kąty obrotu zapisywane w polach `tmpKameraPhi` i `tmpKameraTheta`. W metodzie `RysujScene` wartości te wykorzystywane są jako argumenty funkcji `glRotate`. W momencie zwolnienia przycisku myszy wartości pól `tmpKameraPhi` i `tmpKameraTheta` przepisywane są do pól `KameraPhi` i `KameraTheta`, co oznacza, że obrót jest zapamiętywany.

Projekt 207. Cieniowanie kolorów na powierzchniach

Z trójkątem nie musi być związany tylko jeden kolor. Osobny kolor można związać z każdym jego wierzchołkiem. Wówczas zastosowana zostanie płynna zmiana koloru pomiędzy wierzchołkami (cieniowanie). W niektórych przypadkach możemy cieniowania użyć do imitowania oświetlenia (por. rysunek 11.12).

Rysunek 11.12. *Cieniowanie kolorów może imitować oświetlenie*



1. Modyfikujemy metodę `RysujOstroslup` zgodnie z wyróżnieniami na listingu 11.32.

Listing 11.32. *Z każdym werteksem wiążemy inny kolor*

```
void __fastcall TForm1::RysujOstroslup(float x0,float y0,float z0)
{
   //Rysowanie trojkata
   glBegin(GL_TRIANGLES);

   //ustalanie trzech wierzcholkow trojkata (werteksow (x,y,z))
   //(0,0,z) jest mniej wiecej w srodku ekranu

   //tylna
   glColor3ub(255,255,0); //zolty
   glVertex3f(-x0, -y0, z0); //dolny lewy
   glColor3ub(255,128,0);
```

```
    glVertex3f(x0, -y0, z0); //dolny prawy
    glColor3ub(255,0,0);
    glVertex3f(0, y0, z0); //gorny

    //podstawa
    glColor3ub(0,255,0); //zielony
    glVertex3f(-x0, -y0, z0); //dolny lewy
    glColor3ub(128,255,0);
    glVertex3f(x0, -y0, z0); //dolny prawy
    glColor3ub(255,255,0);
    glVertex3f(0, -y0, 2*z0); //dolny przedni

    //lewa
    glColor3ub(255,0,0); //czerwony
    glVertex3f(-x0, -y0, z0); //dolny lewy
    glColor3ub(255,0,128);
    glVertex3f(0, -y0, 2*z0); //dolny przedni
    glColor3ub(255,0,255);
    glVertex3f(0, y0, z0); //gorny

    //prawa
    glColor3ub(0,0,255); //niebieski
    glVertex3f(x0, -y0, z0); //dolny prawy
    glColor3ub(0,128,255);
    glVertex3f(0, -y0, 2*z0); //dolny przedni
    glColor3ub(0,255,255);
    glVertex3f(0, y0, z0); //gorny

    //koniec rysowania figury
    glEnd();
}
```

2. Kompilujemy aplikację i uruchamiamy ją, a następnie podziwiamy uzyskany efekt (zob. rysunek 11.12).
3. Po tym usuwamy lub komentujemy dodatkowe linie, żeby cieniowanie nie interferowało z oświetleniem figury, które dodamy za chwilę.

Cieniowanie można też wyłączyć poleceniem `glShadeModel(GL_FLAT);`. Wówczas do kolorowania trójkąta używany jest kolor ostatniego wierzchołka. Domyślna wartość włączająca cieniowanie to `GL_SMOOTH`.

Kolor i światło

Co to jest kolor? I jaki jest jego związek ze światłem? Z fizycznego punktu widzenia kolor można identyfikować z długością fali światła docierającego do oka. Podobnie jak w przypadku dźwięku, nasze zmysły dość dobrze radzą sobie z jednoczesną detekcją fal o różnych długościach. Niestety, tak jak w znanym dowcipie o matematyku w balonie, odpowiedź ta jest prawdziwa, dla niektórych fascynująca, ale w przypadku definiowania oświetlenia w programowaniu grafiki trójwymiarowej mało przydatna. Ważniejsza od fizyki okazuje się tu biologia. Mówi ona, że oko rejestruje światło za pomocą trzech

typów czopków reagujących na fale o różnych długościach oraz pręcików, które są czułe jedynie na natężenie światła. Skupmy się na czopkach. Wytworzone przez każdy z typów czopków impulsy nerwowe interpretowane są przez mózg jako kolory czerwony, zielony i niebieski. Pobudzenie wszystkich jednocześnie w równym stopniu, co odpowiada światłu, które jest „mieszaniną" fal o różnych długościach, daje kolor biały (względnie szary, jeżeli pobudzenie jest mniej intensywne). Różne kombinacje pobudzeń czopków prowadzą do całej palety kolorów, jakie szczęśliwie możemy oglądać. Wynika z tego, że kolory czerwony (R), zielony (G) i niebieski (B) tworzą układ współrzędnych (oznaczany symbolem RGB), w którym można zapisać dowolny widziany przez człowieka kolor. Ten fakt został wykorzystany w telewizorze, który generuje obraz zbudowany z czerwonych, zielonych i niebieskich pikseli. W ten sam sposób będziemy również określać kolor źródeł światła.

To jednak nie wyczerpuje tematu barwy i oświetlenia. Do oświetlania używamy zwykle światła o białej barwie. Rzeczy, które oświetlamy, są jednak różnobarwne. Z tego wynika, że czymś innym jest **kolor źródła światła** i **kolor oświetlanych przedmiotów**. To rozróżnienie odzwierciedlone jest również w OpenGL. Kolor oświetlanego przedmiotu, jaki widzimy, zależy wyłącznie od światła, jakie biegnie od tego przedmiotu w kierunku naszego oka. Załóżmy, że przedmiot ten nie emituje światła, a jedynie je odbija. Jeżeli przedmiot ten nie odbija wszystkich długości fali, to część padającego na ten przedmiot światła jest pochłaniana. W efekcie przedmioty oświetlane modyfikują światło, które na nie pada. I to nie znaczy tylko, że je osłabia, ale również zmienia jego kolor. Przedmiot może bowiem pochłaniać różne długości fali w różnym stopniu (innymi słowy niejednakowo odbijać składowe RGB). Zatem jeżeli źródło światła jest białe, a przedmiot pochłania składową niebieską, a dobrze odbija składowe czerwoną i zieloną, to przedmiot ten widzimy jako żółty[13]. I tak właśnie ustala się kolor przedmiotów na scenie w OpenGL — ustalając współczynniki odbicia poszczególnych składowych RGB światła na powierzchni tego przedmiotu.

Żeby bardziej sprawę skomplikować, powiem jeszcze tylko, że funkcja `glColor`, której użyliśmy w projektach 197. i 207., z całą tą teorią nie ma nic wspólnego. Nie zależy bowiem w żaden sposób od oświetlenia sceny — jest zwyczajnym oszustwem, które wprowadzone zostało do OpenGL, aby możliwe było rysowanie bez definiowania źródeł światła. To oszustwo możemy jednak zmniejszyć, jeżeli zażyczymy sobie, aby współczynniki odbicia ustalane zwykle funkcją `glMaterial` były uzgadniane z kolorem ustalonym za pomocą funkcji `glColor` (jednym słowem, aby użycie tych dwóch funkcji było równoważne). Służy do tego funkcja `glColorMaterial`, której użyjemy w kolejnych projektach.

Zacznijmy jednak od możliwości wyłączenia kolorowania ścian. Ułatwi to nam eksperymenty z oświetleniem sceny.

[13] Czasem zamiast mówić, że przedmiot dobrze odbija składowe R i G, wygodniej jest powiedzieć, że odbija składową Y (żółć). Zamiast R i B można podstawić M (magentę), a zamiast G i B — C (cyjan). W ten sposób powstaje nowy układ współrzędnych, oznaczany jako CMY. O ile RGB wygodne jest w urządzeniach opartych na emisji światła (monitory), to CMY bardziej pasuje do określania kolorów na przedmiotach oświetlanych, tj. absorbujących światło (np. wydruk na papierze lub obraz na płótnie). To właśnie kolory z układu współrzędnych CMY uznawane są przez malarzy za kolory podstawowe. Takimi kolorami „plują" także drukarki atramentowe.